AFRL-IF-RS-TR-2004-205
Final Technical Report
July 2004

MULTI-TARGETED PROGRAM GENERATORS

University of Southern California at Marina Del Rey

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D890

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-205 has been reviewed and is approved for publication

APPROVED: /s/

Roger J. Dziegiel, Jr.
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JULY 2004	Final Mar 96 – Jul 03

4. TITLE AND SUBTITLE
MULTI-TARGETED PROGRAM GENERATORS

5. FUNDING NUMBERS
C - F30602-96-2-0192
PE - 63760E
PR - D890
TA - 01
WU - 01

6. AUTHOR(S)
Robert Balzer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California at Marina Del Rey
4676 Admiralty Way
Suite 1001
Marina Del Rey California 90292-6714

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2004-205

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Roger J. Dziegiel, Jr./IFTB/(315) 330-2185/ Roger.Dziegiel@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Although, the benefits of "domain specific" languages and development environments are widely recognized, constructing a design environment for a new domain remains a costly activity, requiring expertise in several areas of software development and the targeted domain.

Elevating system development from the module to the architecture level requires a corresponding elevation in tools for instrumenting, monitoring, and debugging systems. While there is a long history and mature technology for the former, we have just begun to recreate these capabilities at the software architecture level. This report describes two architecture level tools that utilize architecture level instrumentation to monitor software architectures through animation and to create automated drivers for debugging or exercising subsets of those architectures. The latter has been used to give "demonstrations" of distributed systems in which only the user interface is run live by driving that user interface from previously recorded system executions.

14. SUBJECT TERMS
Software Architecture, Software Generators, Instrumented Connectors, Wrapper Composition, Non-Bypassable Mediators

15. NUMBER OF PAGES
41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

1. Overview

This contract was aimed at developing the broad set of technologies required for creating a new generation of code synthesizers, called component generators, that adapt, through regeneration, components to changes in the architecture in which that component operates. Such a capability would allow these generated components to remain compliant as their architecture evolves and to become reusable assets that can be adapted for use in multiple architectures. By eliminating the need to manually revise code generators for each architectural change, this component adaptation technology would remove major cost, expertise, and predictability barriers to the use of synthesis technology in complex systems and place adaptation to an evolving architecture on an equal footing with evolution of the component itself.

The original proposal consisted of a single task to develop architecturally-driven multi-targeted program generators. This research on specification guided retargetable component generation was planned to be carried out using a component generator that was being constructed for DARPA's Advanced Distributed Simulation program. However, when that program was terminated shortly after our contract started, so too was the construction of that component generator.

Moreover, that terminated program was the target client of our architecturally-driven program generators, planning to use our generators in its advanced simulations, and was to jointly fund this task. However its termination occurred before that funding was provided.

With the loss of both the client to utilize the technology to be developed under this task and the funding it was to provide for that development, we focused our effort on instrumenting, monitoring, and debugging software architectures in a task called Instrumented Connectors.

This Instrumented Connector task produced a generic technology for dynamically instrumenting software architectures and monitoring their behavior, and several tools for utilizing this capability. It also allowed mediators to be placed in the connections between components to control that behavior or to integrate those components into larger aggregates, such as expanding PowerPoint into a design editor that allowed external domain-specific design analyzers to interactively track an evolving PowerPoint diagram and provide feedback on its well-formedness.

Because these Instrumented Connectors could be applied to arbitrary programs, without source code access, we recognized that they could be used as a cyber-defense to ensure that executing programs didn’t damage or destroy local resources (such as files or the registry). This recognition led to the addition of an additional task to this contract in 1999 to develop a Non-Bypassable Security Manager for Windows NT.

Each of these tasks is described in the following sections.

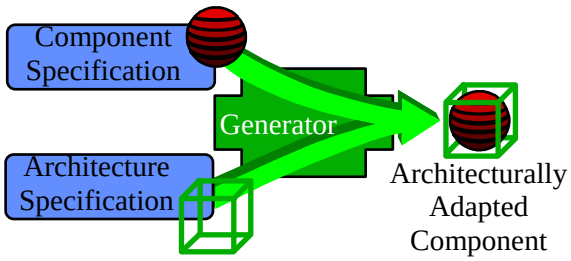
2. Multi-Targeted Program Generators

2.1. Scope

This project is developing the technology for a new generation of code synthesizers, called component generators that adapt through regeneration, components to changes in the architecture in which that component operates. This will allow these generated components to remain compliant as their architecture evolves and to become reusable assets that can be adapted for use in multiple architectures. By eliminating the need to manually revise code generators for each architectural change; this component adaptation technology will remove major cost, expertise, and predictability barriers to the use of synthesis technology in complex systems. This will place adaptation to an evolving architecture on an equal footing with evolution of the component itself.

2.2. Approach

The project will build an infrastructure for constructing a new generation of component generators which accept an explicit description of the target architecture into which the generated component must fit in addition to the specification of the functionality of that generated component. This target architecture specification will be combined with the component specification to produce a generated component for that architecture by merging appropriate "targeting transformations" into the translation process.



These "targeting transformations" will be added to our common "back-end translator" that converts the general purpose executable specifications produced by our "front-end translators" from domain-specific specifications into operational code that uses the interfaces and services defined in the specified target architecture. They will augment the existing set of translation control rules in our "back-end translator" to specify the code sequences needed to invoke these services and utilize the values they return.

This effort will be conducted in three phases. In the first, an existing program generator that uses this meta-program infrastructure, but has wired-in implicit knowledge of its target architecture, will be converted into a target directed component generator by giving it the (limited) set of transformations and pragmatics needed to guide its generation of components targeted to that architecture. Those transformations and pragmatics will then be augmented as needed to guide that same generator to produce components for a second target architecture. Finally, having filled in part of the space for specific (preknown) target architectures, the project will start defining a range of specifiable target architectures and the transformations and pragmatics needed to support this entire range.

These three phases of research on specification guided retargetable component generation will be carried out using a component generator currently being constructed for DARPA's Advanced Distributed Simulation program. That generator converts high level specifications of Semi-Automated Military Forces into simulation modules that fit into the ModSAF simulator and are compliant with its architecture. In the second phase, this generator will be retargeted (via an explicit external architecture specification) to convert those same specifications of Semi-Automated Military Forces into modules that fit into another simulator, the Close Combat Tactical Trainer (CCTT), and are compliant with its architecture.

2.3. Accomplishments

These three phases of research on specification guided retargetable component generation were planned to be carried out using a component generator that was being constructed for DARPA's Advanced Distributed Simulation program. However, when that program was terminated, so too was the construction of that component generator.

Moreover, that terminated program was our target client, planning to use our generators in its advanced simulations, and was to jointly fund this task. However its termination occurred before that funding was provided.

With the loss of both the client to utilize the technology to be developed under this task and the funding it was to provide for that development, we (with DARPA’s agreement) suspended work on this task and focused our effort on the Instrumented Connectors task to instrument, monitor, and debug software architectures.

3. Instrumented Connectors

3.1. Scope

This project is developing the technology to monitor the architectural behavior of legacy systems, mediate those interactions, and architecturally integrate COTS products into larger compositions.

By architectural behavior we mean all interactions between components or with the operational platform. This includes network sockets, RPC, CORBA, OLE2, database access/update, file I/O, terminal I/O, etc.

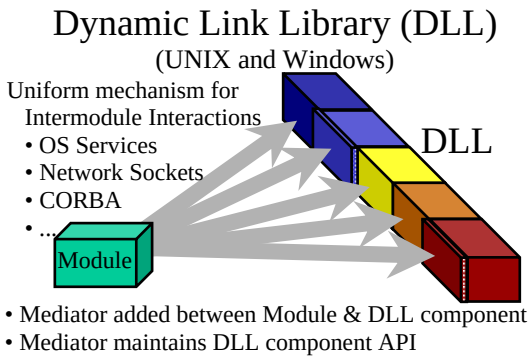
These mediators can be used to instrument architectures, monitor their behavior, integrate legacy components together, or encapsulate potentially harmful or unreliable components

3.2. Approach

Mediators are inserted into the channels through which architectural interactions occur. These mediators are arbitrary programs that can examine the interactions passing through the channel and can throw them away, substitute one or more new interactions, or allow them to proceed unaltered. These mediators can also interact with other mediators to decide what to do and can route the original or altered interactions to additional recipients.

Instrumented Connectors thus have two portions, an interception mechanism that gains access to the interactions occurring within these channels and a mediation program that is inserted into the channel to control those interactions.

The interception mechanism is platform specific and implementations have been developed for both target platforms, UNIX and Windows. The mediators interface to these implementations through a uniform API so that they can be inserted into different types of channels.



The intercept mechanism works by altering the link address of particular entries in a dynamic link library as programs are loaded. The mediators to be inserted in the chosen instrumented connectors are packaged into a newly formed dynamic link library which is linked between the application and the original dynamic link library it intends to use. The mediator thus receives all application interactions through the specified channel and can decide whether to pass them along through the channel, modify them before or after doing so, disallow the interaction, or provide its own response to the interaction.

3.3. Accomplishments

We developed the technology to monitor the architectural behavior of legacy systems, mediate those interactions, and architecturally integrate COTS products into larger compositions. These mediators can be used to instrument architectures, monitor their behavior, integrate legacy components together, or encapsulate potentially harmful or unreliable components.

Intercept mechanism implementations were originally built for SunOS 4.01 and Windows95 without modifying the operating systems or the applications being instrumented. The details of these implementations differ significantly because of the differences in how dynamic link libraries are implemented and accessed on the two systems. The intercept mechanism was subsequently ported to Windows NT, Windows 2000, and Windows XP. We believe this technique will work for other versions of UNIX but have not yet done so.

Mediators are inserted at the time of establishing an interaction channel. Mediator must exist as an executable module on the host platform at the time the channel is established. Choice of which mediator to insert is made by a server (currently centralized) from an architectural specification identifying particular connections to be mediated and the mediator to be used for that connection.

A database (currently centralized) of open channels is maintained as architectural links are formed and broken. An animation tool exists to graphically depict the dynamic architecture connection status. If the mediators also log the message traffic to the central database, then this tool will also depict the message flows through the channels. It can also display and highlight any additional flows introduced by the mediators (especially helpful for demos).

3.3.1. Instrumenting Architectures

Our thesis is simply that the power of the architectural view arises from its focus on the exchange of data and control between components and that for us to effectively design and develop systems at this level; we need tools that provide access to this architectural behavior. This is analogous to the power that our debuggers have given us for subroutine organizations to trace calls to these subroutines, time their execution, and/or insert breakpoints into these calls or their returns. What we therefore need to do is provide similar access to the data and control passing through all the connectors used in any architecture.

Restated in operational terms, our thesis is that **by instrumenting the connectors in architecture, developers can be given access to that architecture's behavior** through a wide variety of architectural tools enabled by that instrumentation.

The problem of course is that we have widened the set of connectors used within our architectures. Furthermore, it appears that each type of connector requires its own instrumentation package that fits within the implementation of that connector. This diversity of connector implementations is exacerbated by the fact that many of these implementations are system provided and not user accessible or modifiable.

3.3.1.1. Instrumented Connector API

We addressed the first of these problems by creating a uniform interface for instrumented connectors so that tools could use the instrumentation data from such connectors, and could control the collection of such data, independently of the type of connector.

These architecture tools receive the instrumentation data through a single stream containing the merged instrumentation from each of the instrumented connectors to which they are attached. Each packet consists of a time-stamp, the sender and receiver of the information, and the transmitted data or an indicator of the connect or disconnect operation that has just occurred on that connection.

These tools determine what data, if any, gets passed on through the connector to the intended recipient(s). They can filter, modify, and/or add extra data to these instrumented communication streams.[1] They can also log this data or copy it to another stream before passing it on.

[1] Because the focus of our architecture research is on monitoring and debugging, the tools reported here don't filter, modify, or augment the communication stream between modules. However, other architecture tools we are building utilize the full Instrumented Connector API.

3.3.1.2. Instrumented Connector Implementations

We addressed the platform specific nature of connector implementations and their user inaccessibility by creating multiple types of implementations of instrumented connectors. The first, externally instrumented connectors, is based on indirection, is platform independent, and doesn't require any changes to the connector implementations being used. However, it only works for connectors for which indirection can be specified (e.g. network sockets and RPC), requires the configuration of the system being instrumented be modified to include these indirections, and doubles the communication cost and number of connectors.

The second type of implementation, internally instrumented connectors, avoids these limitations by changing the platform provided connector implementations. This can only be done when those implementations are accessible (to either users or system administrators). Moreover, the changes are highly platform specific and must be reimplemented on each platform of interest.

Externally Instrumented Connectors

To create a platform independent implementation of instrumented connectors we adopted a strategy of externally augmenting the existing connectors with our instrumentation package. We accomplished this by creating an instrumentation intermediary which sits between two communicating modules and uses separate instances of the connector to interact with each of these two modules. Thus, the connectors shown in **Figure 1** have each been replaced in **Figure 2** by a pair of connectors which connect our instrumentation intermediary with the original communicating modules. As shown in **Figure 2**, each of these instrumentation intermediaries is also connected to some architecture tool which consumes the merged set of instrumentation packets passed to it by these intermediaries and determines which of those packets, possibly modified, should be routed on to the originally intended recipient.

Each of these instrumentation intermediaries consists of two processes which constantly try to read from the two connectors. As soon as data is obtained from one of these connectors, it is passed to the consuming instrumentation tool which processes it and determines

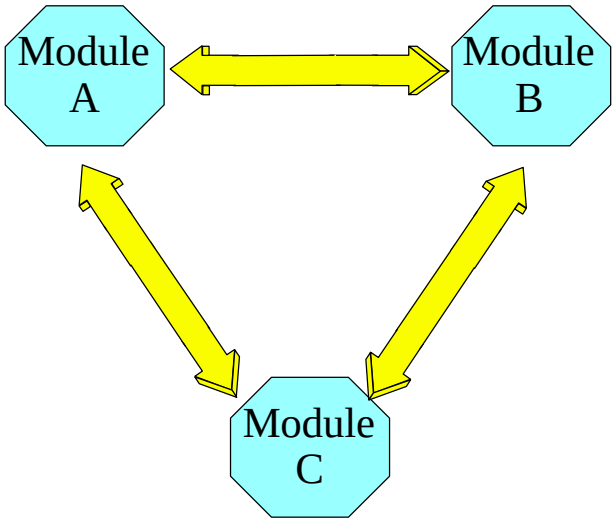


Figure 1: Communicating Modules

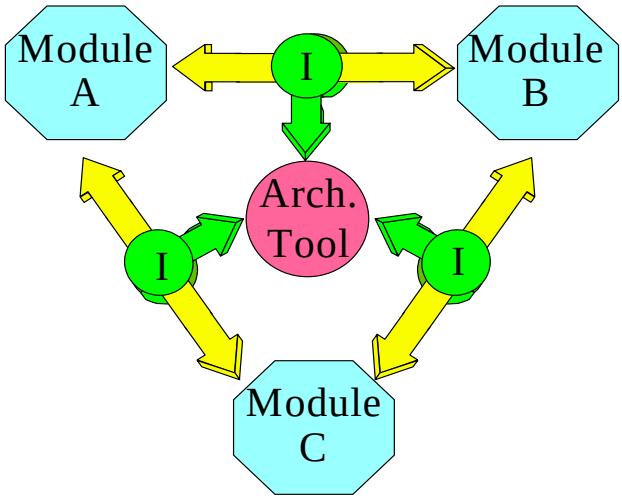


Figure 2: Instrumented Connectors

whether it, or any other data, should be sent to the intended recipient.

This external intermediary strategy has the advantage that it doesn't require access to the internal implementation of connectors and works for many different types of connectors, but introduces the overhead of an extra process and twice the communication, and requires that the intermediaries be explicitly "wired" into the configuration of the system.

Internally Instrumented Connectors

By replacing UNIX's dynamic and static libraries we were able to instrument all of the connectors it provides (e.g. network sockets, RPC, file I/O). The new libraries invoke the architecture tool attached to the connector through the Instrumented Connector API described above in a context in which the communication actions it performs utilize UNIX's original libraries.

Thus, the modified libraries did not have to reimplement UNIX's communication infrastructure. They merely had to allow the appropriate architecture tool to be invoked through the Instrumented Connector API and allow it to utilize the original UNIX communication infrastructure to perform whatever communication actions it chooses to invoke.

This implementation was created for SUN-OS Version 4.3.1 for which source code was available. Other versions of UNIX would require somewhat modified libraries. Other operating systems, such as Windows NT, may require a completely different approach.

These internally instrumented connectors differ from the externally instrumented connectors described in the previous section in an important way. The latter are selectively configured into a system --- only those connectors that the architect wishes to instrument are "reconfigured" to include the desired instrumentation through indirection. However, with externally instrumented connectors **all** instances utilize the modified implementation (the modified libraries), and hence the instrumented implementation.

To provide similar selectivity, a *Selection Table* must be constructed which indicates which connectors to instrument and which architecture tool to attach to those that are instrumented. This selection table is consulted whenever the modified library is invoked (i.e. each time the application invokes a communication action).[2] If the connection is found in the table then the corresponding architecture tool is invoked through the Instrumented Connector API. Otherwise, the requested communication action is simply passed to the original UNIX libraries (i.e. the connection is not instrumented).

Mediator Scope

One can imagine several possible "scopes" over which a mediator might be active:

- all calls to the mediated function(s), regardless of their source
- only calls from designated processes
- only calls from designated threads
- only calls from designated modules (programs or other libraries)

[2] In some operating systems, a dispatch table for the operations on a connection is created when that connection is first formed. By encoding the selection determination into this dispatch table, the selection table in these operating systems only needs to be consulted once, when the connection is formed, rather than on each communication action through the connection.

- only calls from designated trustees (user accounts or user groups)

Currently mediating connectors only support mediation by process scope. The primitive for activating and deactivating requires a set of mediators (known as a *wrapper*) and a process. The same wrapper may be active in multiple processes simultaneously if this primitive has been invoked multiple times. Restricting the mediation by thread or trustee can only be accomplished through conditionality in the mediation code itself.

Wrapper and Mediator Semantics

A *Wrapper* W comprises a set of mediators. Each mediator in the set mediates a distinct function, F, which must be a function exported from some shared library. We use the notation ${}_{W}M_{F}$ to denote "the mediator for function F from wrapper W". The functions mediated by a wrapper need not all come from the same library.

A mediator ${}_{W}M_{F}$ has access to the same parameters as F, and its return value, if any, will be seen as a value returned by F. Within its implementation, a mediator may choose to call F itself one or more times, using the result(s) of the call(s) to compute its own result.

Wrapper Composition

A wrapper implements an enhancement to one or more libraries that will provide new functionality for one or more processes that use the libraries. We want to separate the authoring of wrappers from the ("policy") decision of which enhancement(s) should be applied to which processes on which hosts. This forces us to provide semantics for wrapper *composition*. The description presented in the preceding paragraph was based on a simplistic view that a function has only one mediator.

A wrapper defines one or more *virtual libraries*. Installing a wrapper in a process means forcing the process to use those virtual libraries. We will use the notation L_W to denote "the virtual library L defined by wrapper W". To define wrapper composition, we define how the composition induces a composition of their respective virtual libraries. We will denote the composition of wrappers V and W by $V \triangleright W$. In general wrapper composition is *not* a commutative operation. To emphasize the potential asymmetry, we will read this as "V surrounding W".

When the virtual libraries defined by two wrappers have disjoint base libraries, the composition of the wrappers simply defines the union of the virtual libraries – in this case, the composition *is* commutative.

Two wrappers may both attempt to mediate functions from a common library. That is, wrapper W may contain ${}_{V}M_{F}$ and wrapper V may contain ${}_{W}M_{G}$, where F and G are *different* functions in the *same* library L. Using our "virtual library" metaphor, V and W each define a distinct version of L – L_V and L_W.

Because the virtual libraries are not opaque binary implementations, but rather specifications (described below) of wrappings around selected API's of the base library, it is possible to compose the specifications to produce a single virtual library specification. In the case of mediators for *distinct* functions the composition is trivial. It is just a union of the specifications – the effective virtual library, $L_{(V \triangleright W)}$, contains both ${}_{V}M_{F}$ and ${}_{W}M_{G}$. The composition is again commutative.

But what if wrapper V contains ${}_{V}M_{F}$ and wrapper W contains ${}_{W}M_{F}$? Our solution is to provide a notion of *nested* mediators in a virtual library. In this case $L_{(V \triangleright W)}$ contains a mediator for F that we denote by ${}_{V}M_{F} \triangleright {}_{W}M_{F}$, and read as "${}_{V}M_{F}$ *surrounding* ${}_{W}M_{F}$". The

mediator nesting operator is associative – $(m \triangleright n) \triangleright p$ is the same as $m \triangleright (n \triangleright p)$. We thus find it simplest to understand the following general case. Suppose F is a function from library L, and the effective virtual library L' defined by wrapper composition provides $m_1 \triangleright m_2 \triangleright \ldots \triangleright m_k$ as the mediator for F (each m_l is a non-nested mediator).

- When a process using L' makes an *outer* call (described below) on F, the outermost enabled mediator for F gains control.
- When a mediator m_i executes an *inner* call (described below), the outermost enabled mediator surrounded by m_i in the nesting for F gains control.
- In both cases, if there is no such mediator, the original (unmediated) implementation of F – the one in L itself -- gains control.

Suppose wrappers V and W are composed as described above -- the effective mediator for F is ${}_V M_F \triangleright {}_W M_F$. If both are enabled, an outer call on F will transfer control to ${}_V M_F$. However, if ${}_V M_F$ is disabled, an outer call on F will transfer control to ${}_W M_F$. If both are disabled, an outer call will pass control to F. If ${}_V M_F$ executes an inner call, control will pass to ${}_W M_F$ if it is enabled, or to F otherwise. If ${}_W M_F$ executes an inner call, control will pass to F itself.

Wrapper enablement is a boolean, per-thread, attribute of a wrapper. When we refer to a mediator ${}_W M_F$ being enabled or disabled, in the context of a call on F, we really mean that W is enabled or disabled *in the calling thread.*

Currently, the only mechanism for disabling a wrapper W is to have W's **auto-disable** attribute set. The effect of setting this attribute is that, whenever any mediator ${}_W M_F$ gets control, W will be disabled in the calling thread. It will be re-enabled when ${}_W M_F$ returns. The setting of **auto-disable** is part of a wrapper's definition and cannot be changed dynamically. As a consequence, a thread's calling stack will *never* contain frames for both ${}_W M_F$ and ${}_W M_G$ simultaneously, nor will it ever contain multiple frames for ${}_W M_F$.[3] If you specify **auto-disable** for a wrapper W, then you are allowing *all* calls that occur dynamically within *any* mediator of W to go unmediated by W.

Outer Calls and Inner Calls

All calls on a function F that occur dynamically outside the implementation of any mediator of F are termed *outer* calls. A non-composite mediator $m = {}_W M_F$ has access to two "versions" of F that it may (but need not) call:

- The implementation seen by non-mediators. Calls on this version are *outer* calls. If ${}_W M_F$ happens to be the *only* (enabled) mediator for F, and if W is *auto-disabling*, then an outer call executes F's original implementation.
- The inner version relative to ${}_W M_F$. Calls on this version are termed *inner* calls. If m itself is the effective mediator for F, or is the innermost mediator of a composite mediator $m_1 \triangleright m_2 \triangleright \ldots \triangleright m$, the inner version is F's implementation in the original library. If the effective mediator for F is $m_1 \triangleright m_2 \triangleright \ldots \triangleright m \triangleright m_i \triangleright \ldots \triangleright m_k$, then the inner mediator is $m_i \triangleright \ldots \triangleright m_k$.

[3] This behavior is not sensible in general for threads which contain multiple fibers, where wrapper enablement should be a per-fiber, not a per-thread, attribute. However, if neither the mediators nor the mediated function itself actually switch fibers, the behavior is the same as if enablement were really tracked on a per-fiber basis.

Note that *neither* of these versions is necessarily, but that either may be, the *original* implementation of F from its library. Also note that, if W is not *auto-disabling*, then even an inner call may lead to a recursive entry of ${}_{W}M_{F}$. This can occur if an inner mediator, or even the original definition of F, performs an outer call on F.

Figure 3 depicts a function F with three mediators, and shows the entry points for an outer call and for an inner call from the middle mediator. The figure assumes all three are enabled.

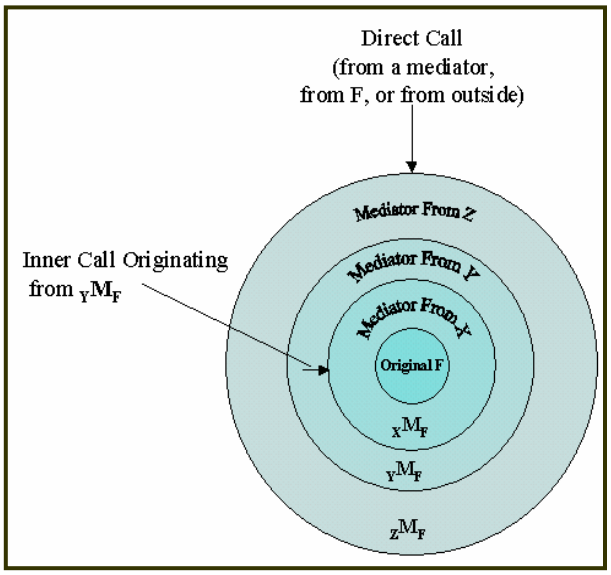


Figure 3: Function with 3 mediators

Most primitive mediators will, at least conditionally, need to make an outer or inner call to compute their result. As the above analysis indicates, the two kinds of call may have different results, depending on:

- Whether the mediator's wrapper has the **auto-disable** attribute. This is controlled by the wrapper's author, so it is known when the mediator is coded.
- Whether the mediator has been composed with other mediators. This is generally not known when the mediator is coded. Currently, no runtime utilities are provided to query a mediator's position in a composition, so it is not possible to write a mediator which chooses between outer and inner calls at run time.

Fortunately, a few rules of thumb can help sort out this complexity, at least in the case where a wrapper is intended to compose with as-yet-unknown other wrappers.

- A mediator ${}_{W}M_{F}$ should make an outer call if, and only if, (a) the call is passing different parameters than those passed to the mediator, or (b) prior to making the outer call the mediator has changed global state that is documented as relevant to F's result. Making an inner call in these cases would deprive any outer mediators of an opportunity to mediate the revised call.
- An outer call should only be made conditionally. A mediator must ensure that, at least in the absence of other mediators, it can break the recursion created by outer calls.

- Use of the **auto-disable** attribute implies knowledge about composition, because the mediators of such a wrapper will be unable to mediate outer calls originating in wrappers they surround. When **auto-disable** is omitted from a wrapper's specification, however, each mediator ${}_{w}M_{F}$ must ensure that it does not initiate a calling sequence that will lead to a recursive outer call of F with identical parameters and state. In particular, a cycle could result from calling some function other than F that leads to an outer call on F. This is rarely a problem, but if it is unavoidable ${}_{w}M_{F}$ must use thread-local state to detect and break the recursion.

  Not all applications of wrappers, however, require building wrappers that can compose with others. In such cases, there are both conceptual and performance advantages to using the **auto-disable** attribute.

Mediators that Fail Intentionally

Functions in Windows NT libraries communicate their results back to a caller in one or more of the following ways:

- a single return value
- modification of parameters
- modification of global or thread-local state

If a mediator ${}_{w}M_{F}$ fails to use the same conventions for returning results as does F itself, it is likely to cause an error in the calling process or an outer mediator. This is of particular concern when ${}_{w}M_{F}$ wishes to "fail" even though F, when called in the same context and with the same parameters, would not fail. Windows NT provides an implicit thread-local integer *error value* variable to every process. Many API's which can "fail" indicate failure by both (a) returning a designated value (typically NULL) that is never returned in a successful call, and (b) setting the error variable to provide the caller with more detail about the nature of the failure. A mediator that wants to indicate failure should be implemented to set this variable as well as return the designated value. Usually one of the documented values for the error value will be suitable for the mediator's purposes – meaning that the calling process should be prepared to deal with that error from F itself.

Wrapper Definition

Wrappers are defined in ASCII files. Although we conventionally give these files ".wrp" file type, the software neither relies on or defaults to that type.

A wrapper definition looks like:

```
wrapper name implementation impdll [properties]
  wrap APIname in dll with mediator size integer
  ...
  wrap APIname in dll with mediator size integer
```

name is an arbitrary name chosen for the wrapper. Currently, this name is used only to designate a wrapper to remove from a process, and to ensure that each wrapper installed in a process has a distinct name. For these purposes, the name is case-sensitive. The name will play a more prominent role when site administration facilities are added. *impdll* is the library which implements the wrapper.

Each wrapper has two boolean-valued properties. There are four keywords that may be used in *properties* to specify these properties.

- **propagate** or **no_propagate** specifies whether the wrapper is self-propagating. Self-propagating wrappers are discussed in Wrapper Propagation. If neither keyword is present in *attributes*, the wrapper will be self-propagating.

- **auto_disable** or **no_disable** specifies whether the wrapper is *auto-disabling***.** See Wrapper and Mediator Semantics for details. If both keywords are omitted from *attributes*, the wrapper will not be auto-disabling.

Each **wrap** directive identifies one mediator for the wrapper. *dll* is the library containing the function to mediate, and *APIname* is the function's name. *mediator* is the name of the function exported from *impdll* that is to mediate *APIname*. Finally, *integer* is the number of bytes of parameter expected by the function named by *APIname*.

Wrapper Restrictions

No wrapper may specify multiple mediators for a single API.

Each wrapper installed in a process at any one time must be implemented by a distinct library.

Each function used as a mediator must be exported from the library under the name used in the wrapper definition.

Mediator Performance

The technique used to implement mediators imposes some overhead on each call to a mediated function – the linkage from the caller to the mediator is not a simple "call" or (indirect) "jump". This overhead is independent of the number of mediators placed on an API and also independent of the API being mediated, except for a small amount of code that copies parameters and is proportional to the number of bytes of parameter. The overhead involves the algorithm depicted in **Figure 4**.

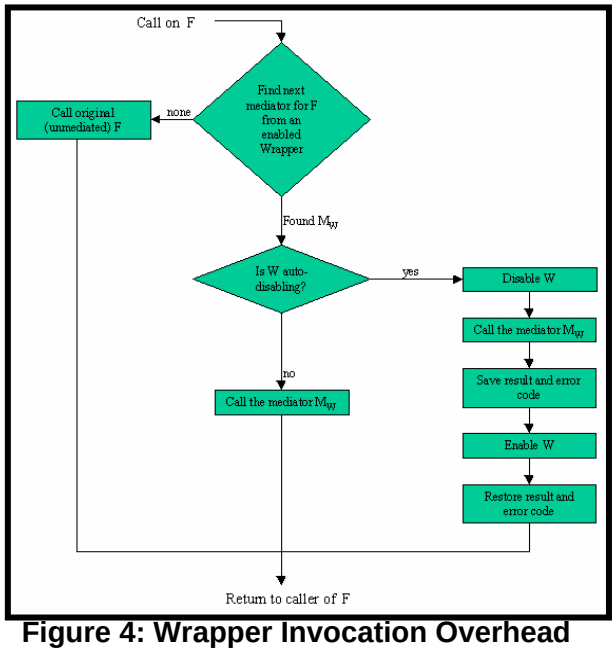


Figure 4: Wrapper Invocation Overhead

Whenever an outer call is made on a mediated API F, or whenever a mediator for F issues an inner call, the mediation runtime manager will find the next enabled mediator to be invoked. If no mediators remain to be called, the original implementation of F is invoked. If a mediator is found and its wrapper is not *auto-disabling*, the mediator is simply invoked. Otherwise, the wrapper is marked as *disabled* during the period the mediator is active.

It is clear from the flowchart that the cost of running ${}_{W}M_{F}$ is increased if W is marked as *auto-disabling***.** However, the additional overhead will be more than reclaimed in savings *if* ${}_{W}M_{F}$ makes any *outer* calls on F or any calls on other functions being mediated by W, because each such call will be able to completely bypass the execution of its mediator in W.

There are situations where an outer and an inner call are equivalent, such as when the *only* mediator of a function belongs to an auto-disabling wrapper. In such cases, the inner call is slightly, but almost certainly negligibly, more efficient.

Parametric Wrappers

It is possible to write a wrapper whose behavior is parameterized by values established when the wrapper is installed in a process. For example, a wrapper that limited use of some shared resource might allow a numerical limit to be provided at wrapper installation time rather than at wrapper definition time.

A wrapper parameter's value is determined by a process installing a wrapper and consumed by the process in which the wrapper is installed. These are almost always *distinct* processes. For that reason, the parameter value may not be, or contain, pointers from the installing process' address space. A wrapper parameter value is simply an arbitrary size block of data whose first four bytes contain that size. The *interpretation* of the data is a matter on which the wrapper and the installer must agree.

Wrapper Propagation

When a process with installed wrappers spawns a new process (via one of Window's **CreateProcess** APIs), any self-propagating wrappers from the spawning process will be installed in the new process as well. The wrapper parameter is propagated as well.

The method by which wrappers are propagated ensures that they are installed in the new process before that process's main thread begins execution.

Propagated wrappers will have the same nesting relationships in the new process as they had in the spawning process.

No wrapper state is propagated to the new process. Removal of a wrapper from a process has no affect on its presence in processes to which, or from which, the wrapper was propagated.

Wrapper Installation Atomicity

The installation primitive for installing a wrapper executes within the process being mediated. Correct operation of the mediators may depend on *all* of them being installed. If other threads of the mediated process are active while the installation is being carried out, those threads might execute in a state in which some, but not all, of the functions being mediated have had their calls rerouted. For GUI-based applications this is

generally not a problem, since the mediators can be installed while all threads are blocked waiting for user input.

3.3.2. Architecture Animator

The Architecture Animator displays the architecture of an instrumented system as a graph in which the components are nodes labeled by the component name and the connectors are links between these nodes. As the connections between the components are made and broken the graphic depiction of the connector is respectively displayed and erased. As data is passed through the connector a short identifying label is displayed along the connector together with an arrow indicating the direction that data is flowing through the connector.

The Architecture Animator works by sequentially processing connector instrumentation packets. These may either be packets coming from a live execution of the instrumented system or previously recorded packets stored in a file. The state change captured in each instrumentation packet is displayed on the architecture graph and the animator moves on to the next packet. A speed parameter allows the user to slow the display update down to an acceptable rate or to temporarily halt further screen updates.

3.3.3. Architecture Driver

Like the Architecture Animator, the Architecture Driver also sequentially accesses connector packets. However, unlike the Architecture Animator the objective is not updating the architecture's state on a display for the user, but rather to exercise the live execution of some subset of the instrumented system by simulating the behavior of the rest of that system.

Consider a simple system consisting of the three modules shown in **Figure 5**. Each module has a single connector to each of the other two modules through which it communicates with them. Finally, assume that Module A is to be exercised by the Architecture Animator because a new version has been produced which we want to unite and integration test with the rest of the system, because we need to examine it in a controlled environment, or because that module displays interesting behavior that can be used as a "demonstration" of the entire system (if properly driven by a simulated environment).

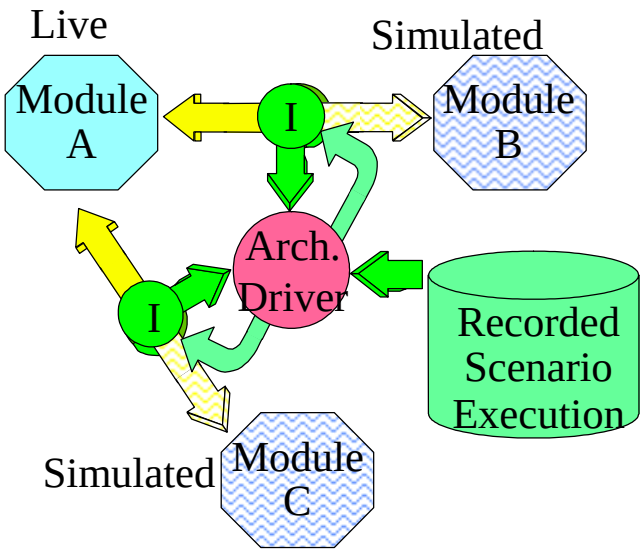


Figure 5: Simulated Interaction

As shown in **Figure 5** the live modules (Module A in this example) are run with instrumented connectors that funnel all the output of the live modules to the Architecture Animator. These outputs are not passed on to their intended recipients (here Modules B and C) through the instrumented connector as they normally would because those modules are not actually being run, but instead are being simulated by the Architecture Animator. The Architecture Animator also sends the outputs of the simulated modules, which it obtains from the recorded scenario, that are intended for the live modules through the instrumented connectors.

These packets are always read from a previously recorded instrumented execution of the system which contains the behavior of each module in that system. The Architecture Animator sequentially sifts through the recorded instrumentation packets to find all the ones that contain either communications intended for, or communications coming from, one of the live modules. The former are used to provide the inputs needed by the live modules as they execute, and the latter are used to track the progress of those live modules through the recorded scenario to determine when it is appropriate to provide them with those inputs.

This progress tracking is central to the Architecture Animator's ability to create an accurate simulation for the live modules. It rests upon monitoring the outputs of the live modules (provided by the instrumented connectors) and correlating them with instrumentation packets in the previously recorded instrumented execution of that system.

When an instrumentation packet is found which contains communication from a simulated module to one of the live modules, it is immediately sent to the corresponding live module through the appropriate instrumented connector. The Architecture Animator then continues with the next instrumentation packet that either contains communication for, or communication from, a live module. The former are processed as just described.

However, when the Architecture Animator reaches an instrumentation packet which contains output from a live module, then it stops processing further information packets for that connector until the *corresponding* output from the live module is received through the appropriate instrumented connector. Correspondence between the recorded and live

scenarios is defined by the sequence index of those outputs in each connector. That is, the nth recorded output from a live module through a connector corresponds to the nth live output from that module through the same connector. Thus, the Architecture Animator demands that the temporal ordering within a single connector remain constant. However, it allows the ordering between connectors to be non-deterministic.

This non-determinism between the ordering of outputs received through different connectors reflects the inherent asynchrony of distributed systems, or even concurrent processes interleaved on a uniprocessor. The Architecture Animator contends with this asynchrony by maintaining separate *reader* processes for each connection with the live module(s). Each such reader process is in an infinite read-then-queue loop in which it repeatedly does a blocking-read for data being output by the live module and then FIFO queues that data on the *received-inputs* queue associated with that connection.

When the main Architecture Animator process, which is sifting through the information packets in the recorded scenario execution encounters an output from a live module, it examines the received-inputs queue associated with the corresponding connection. If it is non-empty it removes and treats it as the output that corresponds to the output just obtained from the recorded scenario execution. If the received-inputs queue is empty, the Architecture Animator waits for the live module output it needs to continue its progress through the recorded scenario execution. Thus, the Architecture Animator will slow down and wait for anticipated outputs from the live module(s), and will thus ensure that those anticipated outputs are not altered by the premature arrival of simulated inputs that occurred subsequently in the recorded scenario (because it will not process any subsequent portions of that recorded scenario until the anticipated output is received).

In addition to time synchronizing the simulated inputs to the execution speed of the live module(s) by maintaining a correspondence between the outputs of the live module(s) and the recorded scenario, the Architecture Animator also checks whether these corresponding outputs are *equivalent.* If so, then the live module is behaving as "predicted" by the recorded scenario. Otherwise, the live module has deviated from the recorded scenario and the Architecture Animator signals an "off-scenario" exception.

The Architecture Animator allows both the definition of equivalence and the handling of off-scenario exceptions to be user-defined. The former is accomplished through a user supplied predicate which compares the recorded and live outputs and determines whether they should be considered equivalent. The latter is accomplished by giving control to a user supplied routine which can provide application specific responses (i.e. simulated inputs to the live module), indicate when and where the live and recorded scenarios have been resynchronized (if ever), or cause a break.

It also allows the user to supply a routine which selects the recorded execution to use for driving the live module(s). This capability is necessary if this determination is to be made dynamically during the execution of the live module(s), rather than before they start execution.

3.3.3.1. AEGIS Example

The DARPA ProtoTech community has created a next generation prototype of a portion of the Aegis system which tracks and responds to predicted intersections of hostile and friendly aircraft with three dimensional geometric regions attached to ships or fixed to the surface of the earth. The responses include issuing friend-or-foe challenges, concluding that an aircraft is hostile and launching missiles against hostile aircraft.

The prototype consists of five modules, each produced in a different prototyping language developed within DARPA's ProtoTech program, a Track Server which provides radar track data (position, course, speed, acceleration, etc.), and a Testbed Controller which allows users to construct scenarios, control their execution, monitor the operation of the prototype, and display intermediate results.

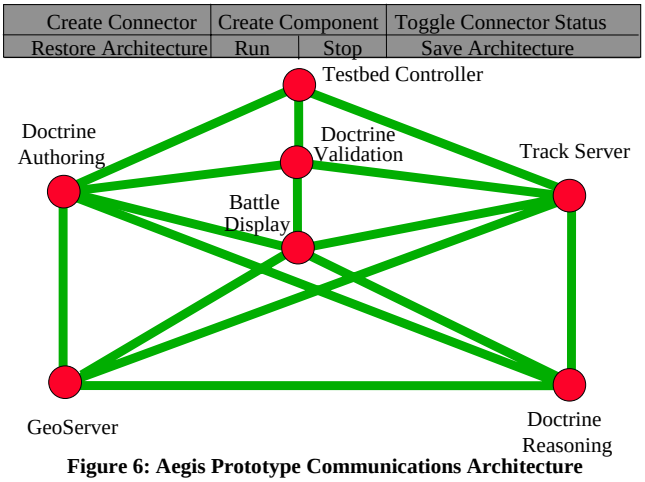


Figure 6: Aegis Prototype Communications Architecture

The communication architecture for this prototype is shown in **Figure 6**. All of these inter-module connections are via UNIX network sockets.

Using our externally instrumented connectors implementation of network sockets, we recorded the execution of several different scenarios. We then used the Architecture Animator to display the architecture level behavior of the prototype on these scenarios. **Figure 7** is one screen snapshot showing which connections were established at that instant and what the most recent message was along each of those established connectors together with its flow direction.

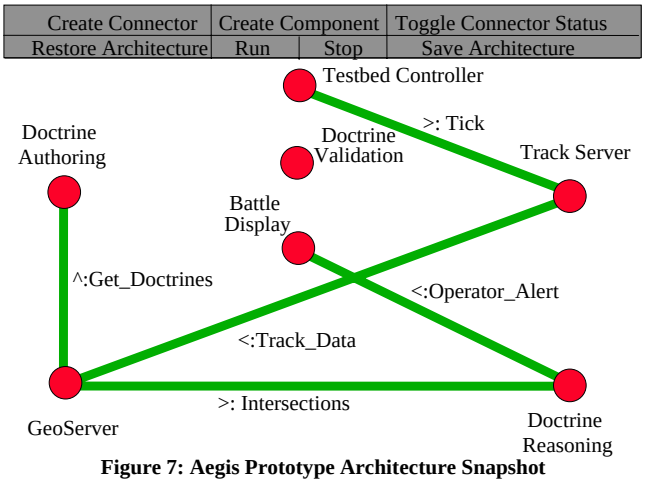


Figure 7: Aegis Prototype Architecture Snapshot

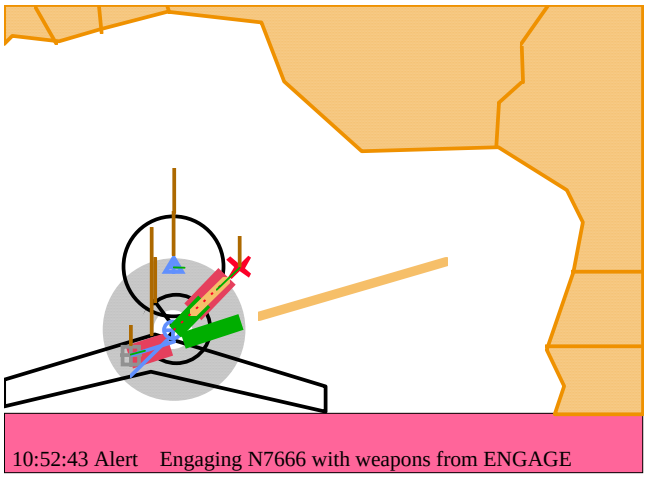


Figure 8: Snapshot of Testbed Controller Screen

Finally, **Figure 8** is a snapshot of the Testbed Controller's screen, taken during its live execution in the simulated environment created by the Architecture Driver, which shows the position, course (direction of the green line), speed (length of the green line), and altitude (length of vertical tan line) of friendly (blue) and hostile (red) aircraft, their predicted intersections (colored line segments lying along the aircraft's path) with fixed (polygon) and moving (donut and circle-minus-wedge) geometric regions, the predicted missile firing and engagements points (the head and tail of the dotted pink lines), and the operator messages issued (along the bottom of the display).

The important point is that this display is exactly as it would appear in a live execution of the entire Aegis prototype, even though the Testbed Controller is the only module actually executing and its display combines the Testbed Controller's local information with data computed by the simulated modules (the predicted intersections provided by the GeoServer and the predicted missile firing and engagements points and operator messages provided by the Doctrine Reasoning module).

Moreover, once the internally instrumented connectors implementation is used (when the selection table implementation is complete), the Testbed Controller can be used completely unmodified. However, because UNIX sockets are the only type of connector we have instrumented in the externally instrumented connectors implementation currently being used, we had to make one change to the Testbed Controller so that the Architecture Driver could simulate the execution of whatever scenario the user dynamically selected through the Testbed Controller's interface. This interaction with the user occurred through an X-window menu selection and resulting in opening the selected scenario file, both of which are invisible to the Architecture Driver since it doesn't have instrumented connectors through which it can monitor these communications. We therefore modified the Testbed Controller to write a file containing the name of the user selected scenario. The user-supplied recorded instrumentation selector (one of the Architecture Driver's parameters) used the contents of this file to identify the recorded execution corresponding to this selection.

3.3.4. Application Program Interface (API) SPY (Smiley)

We implemented an API Spy that enables users to monitor intermodule API interfaces, including parameters and return values, to understand the interactions between those modules. Users can interactively choose which functions in which libraries to monitor,

how to format the recording of those function calls, and the conditions under which to cause a break. This tool is our primary means of determining which interfaces to mediate when creating a new wrapper.

We also created a Mediation Toolkit to simplify the use of this mediation technology by other developers so that they can create and install their own mediators. The toolkit consists of the API Spy, the mediation installer and wrapper propagator and a set of macros for invoking the original interface being mediated and catching and signaling exceptions.

Smiley is an interactive utility that allows an analyst to log information about calls on functions exported from dlls. The logging takes place only for processes, libraries, and functions designated by the analyst. The analyst determines the content of the logged information (e.g., which, if any, parameters to include) and can set breakpoints on selected calls, transferring to a debugger to explore a program's state, or single-step the program's execution, when a breakpoint is reached.

Through its interactive graphic interface, the analyst *opens* one of the displayed running processes by left-clicking its name. Smiley then lists, indented below the process name, the names of all shared libraries currently in use by that process. The analyst left-clicks a function name within an open library to select that function for monitoring. A camera icon appears next to the name to indicate that it is being monitored. The analyst may also select or unselect groups of functions to monitor (by their link status (static or dynamic) or by name matching).

Having selected one or more functions from one or more libraries of a process for monitoring, the analyst clicks the command button labeled "Monitor". Monitors for the selected functions are then installed in the running process.

A Smiley Trace window for the process then appears. The window contains a scrollable text display and four command buttons. Although the monitors were installed by the "Monitor" command, they are inactive. (The process itself may be running, but the monitors are not reporting calls.) To activate the monitors the analyst clicks the "Resume" button in the trace window. As calls on the monitored functions occur in the monitored process, a trace appears. Each call produces a single line of text, displaying the name of the function and the thread id of the calling thread. Indentation is used to help visualize nested calls.

At any time, the analyst may click the trace window's Pause button to deactivate the monitors in a process. He can later reactivate them with the Resume button. When monitoring is paused, the analyst is able to scroll through the (often voluminous) trace information in the text window, or use "copy and paste" to copy the trace text to another program.

To remove the monitors from a process entirely, the analyst closes the trace window using the standard Close button. He may do this while the monitors are inactive or active.

When a process invokes a monitored function, a report is generated for the process's Smiley trace window. Because transmission of the report to the Smiley process occurs asynchronously, there may be a considerable latency in the appearance of reports. They always appear in the order in which the calls occurred, however.

3.3.4.1. Adjusting the Set of Monitored Functions

If the analyst is dissatisfied with the set of functions being monitored, he can simply remove the monitors (by closing the trace window), modify his selections in the Smiley Control GUI, and click the "Monitor" button to install the updated selection.

Frequently, however, he finds that the trace window is being flooded with calls on one or two of the selected functions, and he would simply like to deselect those. This is accommodated directly from the Smiley trace window. With monitoring paused, the analyst selects trace lines from one or more of the monitored functions. He then clicks the "Unmonitor Selected APIs" button to remove those functions from the monitoring set, and reactivates the remaining monitors with the "Resume" command.

3.3.4.2. Reporting Parameter and Return Values

Although the order, nesting, and threads of function calls can provide important insight into a program's implementation, far more can be learned if the log contains the parameter and result values of the calls as well. To obtain such augmented traces, the analyst makes use of Smiley's "Monitor Tailoring" pane.

When a function is selected in the Process Composition pane, the Monitor Tailoring pane displays that function's prototype – the order of parameters, their types, and the return type, if any. The Monitor Tailoring pane also provides two text boxes in which the analyst specifies the content of trace output he would like reported on entry to and exit from calls on that function.

In the trace output specifications, the analyst can include both literal text and parameter values. Parameter values are referred to with the notation **%n%**, where **n** is the index of the parameter whose value is to be printed in the output. In the "after" box, the analyst can also refer to the value returned from the function call using the notation **%0%**.

Other notations in the trace output specifications allow the analyst more control over the formatting of the parameter and return values, such as printing in decimal or hex, or printing the value of fields of structures pointed to by parameters or results.

3.3.4.3. Breakpoints

In addition to the text boxes for before/after trace output specification, Smiley's Monitor Tailoring pane has two checkboxes. These are used to set breakpoints at the entry and/or exit of a monitored function. A breakpoint can be set regardless of whether any corresponding trace output is specified. When a breakpoint is reached, control is transferred to the analyst's preferred debugger. Precisely what can be done at that point is debugger-specific. With the Microsoft Visual Studio debugger, for example, the analyst can follow pointer paths from the function's parameters and can step through an assembly language level representation of the code. Using the debugger's "resume" command, execution continues with all Smiley's monitors and breakpoints in place.

3.3.4.4. Persistence

Having selected a set of functions to monitor, and possibly tailored some of those monitors as just described, an analyst often wishes to perform the same monitoring on multiple runs of a program, or even on other programs. The analyst can give a name to such a configuration of monitors and register the configuration through Smiley's

configuration management pane. The same pane allows the analyst to view the names of registered configurations and restore one of them to a selected process.

Configurations are registered on a per-library basis That is, a named configuration contains only monitoring selections for a single library. To save and restore a configuration in which functions from several libraries are monitored, the analyst must save and restore a configuration for each library involved.

3.3.5. COTS Integration

We used our instrumented connector technology to integrate several COTS products with each other or into larger aggregations with third-party tools. The most extensive of these was the extension of Microsoft PowerPoint into an architectural editor, but we also integrated EMACS as the message composition editor for Eudora, extended Internet Explorer into a Personal Web Annotator, and also extended it into an Ad blocker.

Each of these COTS integrations is described in the following subsections.

3.3.5.1. Architecture Editor (from PowerPoint)

Summary

PowerPoint has been extended into an architecture editor (called the Design Editor) by instrumenting its connection to its user interface. As architecture diagrams are constructed and modified in PowerPoint, a logical database of the evolving architecture is dynamically maintained. Each change to the architecture is analyzed in real time and static analysis errors are graphically depicted as annotations on the diagram. This integration was accomplished without any changes to PowerPoint.

Motivation

Domain-specific languages and development environments are frequently proposed as a means to improve the productivity of designers. Although prototypes of such languages and environments proliferate in conference proceedings, commercially viable examples remain rare. We believe that the reason for this is primarily the difficulty of implementing, not of designing, a high-quality design environment for a new domain.

There are two major parts of a domain-specific design environment for an engineering domain. The first is a graphic user interface that lets an engineer intuitively manipulate the objects constituting a design, create reusable sub-designs, and navigate within and between designs. The second is an integrated toolset that provides the engineer with feedback on a design – problems, metrics, scenario animations, etc.

We believe that the first portion – the GUI – requires only shallow knowledge of the application domain on the part of the environment builder. The second problem, although it may have graphical presentation aspects, relies on a much deeper understanding of the domain.

Implementation

The *Design Editor* generator addresses these two areas in disparate ways. It simplifies the GUI-building task by extending a high-quality commercial, but nondomain-specific, platform for constructing and presenting graphics – Microsoft PowerPoint – rather than

some lower-level graphic library such as Motif or GUI constructors such as VisualWorks or Visual Basic. The generator's "specify by example" paradigm casts the creation of the GUI for a new domain as a graphical task in its own right, rather than a programming task. PowerPoint itself provides the preponderance of the design editing GUI, which is common across engineering domains.

The design environment generator provides a flexible runtime architecture for incorporating feedback programs (called analyzers) into the generated environments. These analyzers can be written in the programming language, and run on the machine, of the implementer's choosing. The communication protocols used by the analyzers and the design editor allow analyzers to be written using either batch-oriented or incremental algorithms. This flexibility should make it relatively easy to import preexisting domain-specific feedback programs into the generated design editor environments.

The analyzer-editor protocols also support common graphical presentation requirements of feedback, permitting the design editor to reflect analyzers' results directly onto a graphical design, rather than requiring an analyzer to provide its own GUI for that purpose.

Figure 9 shows the roles of the domain expert, the analysis programmers, and the GUI designer in producing a domain-specific design environment for engineers.

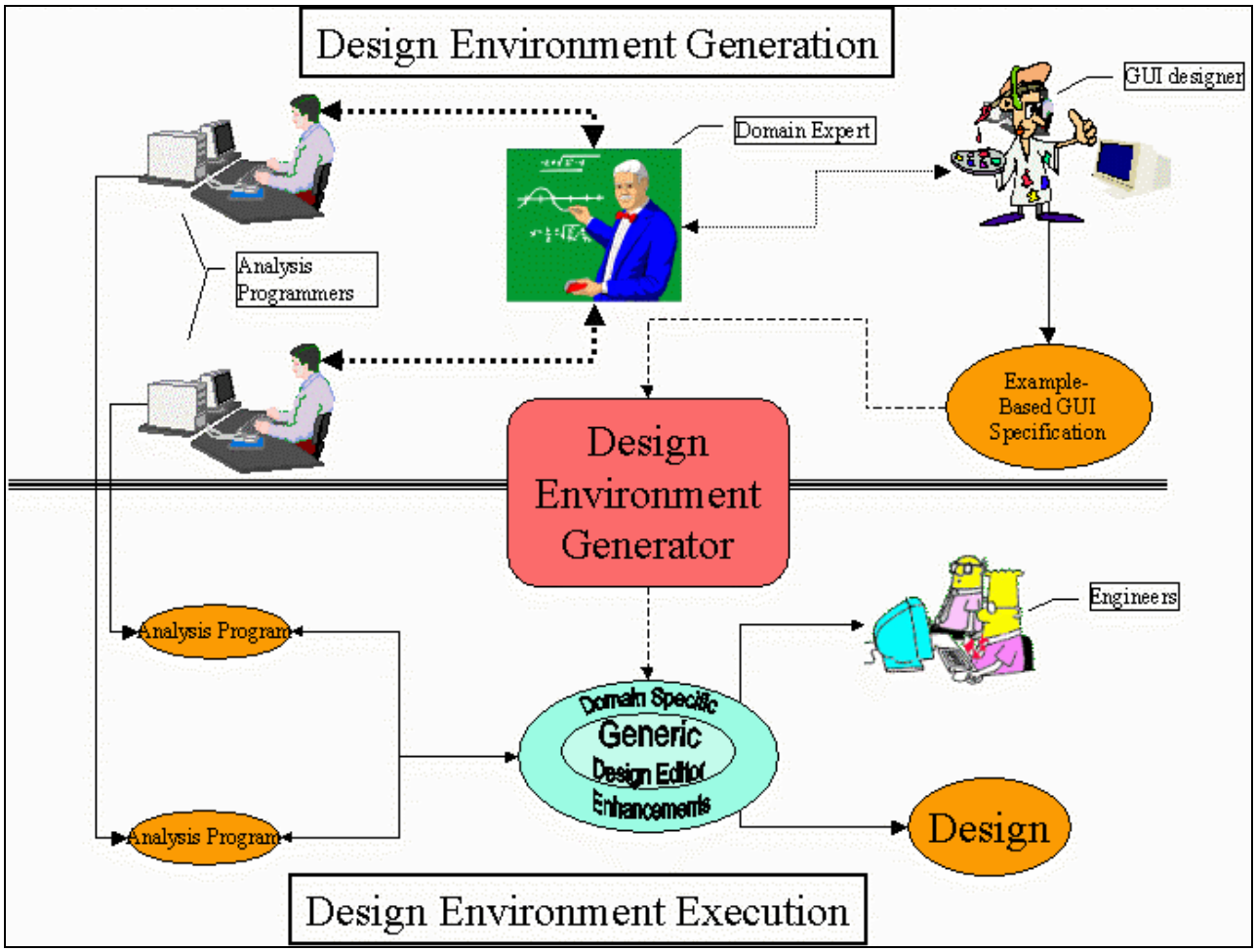


Figure 9: Design Environment Generation

Domains and designs

Common to numerous engineering domains is the "box-and-arrow" character of visual designs. Boxes represent *components* of a design artifact. Each component denotes an instance of some *component type* – resistors and capacitors, tasks and workers, cargoes and vehicles – the types used are highly domain specific. Each arrow represents a relationship between the components at the ends of the arrow. These relationships may be physical, temporal, or neither. A single design may reflect several different *relationship types* – such as control flow and data flow in a software algorithm. In most domains, not all instances of a given component type are identical. So the types are parameterized by *properties* — such as the capacity of a storage tank or the power of a lens. Like components, relationships may also have properties – such as the gauge of a wire or the delay of a communications link. We currently support properties with boolean, integer, real, and string values, as well as with values from domain-specific enumerated types. A property value may consist of a single value from one of these types or a set of values.

The *units* of a design are the component and relationship instances in the design. Knowing the unit types of a domain and the properties of each constitutes the shallow syntactic knowledge of the domain. By itself, it is not sufficient to produce a semantically meaningful, much less useful, design.

Nevertheless, this shallow knowledge is significant because *this is the level of information exhibited in graphic designs*. This simply reflects the fact that this level of representation is sufficient for two crucial purposes:

- Engineers (or software) with a deeper understanding of the domain can *derive* the information they need from it. It thus serves as the basis for analysis and communication between engineers.
- Other people (or software) can construct artifacts from it (i.e. implement the specified design) *without the need for a deeper understanding of the domain*.

Our work focuses on leveraging the central role of these shallow domain models within a design environment. Our contributions are:

1. Generate a domain-specific design editor for a domain without any traditional programming
2. Provide a framework for analysis programs to track an evolving design and provide feedback.
3. Generate the domain model for these analysis programs

Analyzers and Analyses

Although graphical designs are often used solely for their value for human visualization and communication, they become more valuable if software tools can also provide analyses and/or implementations of a design. Informally, we consider an analysis to be any body of information derived from a design. Examples of analyses are:

- Design correctness feedback
- Cost and performance analyses
- Automatically generated implementations of software designs
- Animating a usage scenario on a design

Each domain has its own idiosyncratic analyses, whose requirements for design data, synchronization, and feedback mechanisms may vary substantially. To accommodate these variations the design environment architecture allows analyzers to be independent components that communicate with the editor through an object-oriented protocol for exchanging design information and analysis feedback. The design editor provides an analyzer with incremental updates to the design state. An analyzer may also query the editor to find out about particular aspects of the design state. This allows a variety of implementation techniques to be used in analyzers.

An analysis may be parameterized. The parameters of an analysis are just like the properties of a design unit, with one exception. An analysis may be "focused". What this means is that it has a parameter consisting of a set of units from the design being analyzed. An analyzer will typically use this focus parameter to restrict its analysis to the portion of the design designated by the focus set.

Analyzers execute as separate processes, possibly not even on the same machine as the design editor itself. The relative independence of analyzers means that an analyzer could implement its own GUI for presenting analysis results to a designer. However, to simplify the implementation of analyzers, and provide for graphical presentation of feedback on the design itself, analyzers *may* make use of a predefined reporting mechanism in the analyzer-editor protocol.

An analyzer may send the editor an *analysis* consisting of one or more *result*s. Each result consists of a textual *explanation* together with a (possibly empty) set of *markups*. The markups provide graphic feedback to augment the explanation. Each markup can specify:

- that a unit be highlighted
- that a unit be hidden
- that a component port or arrow terminus be labeled with specified text.

For example, a report might have the explanation "Only one input is allowed at the control port of a thermostat." The accompanying markups might call for highlighting two arrows terminated at the same control port of a thermostat, and labeling that port with the text "too many inputs".

We divide analyses into two categories: *snapshot* analyses and *incremental* analyses. An incremental analyzer that uses the report/markup mechanism for presenting feedback is expected to update the analysis each time that it receives an update to the design state.

Updates to the design state are actually grouped into transactions in the editor-analyzer protocol. Incremental analysis updates are expected to follow each transaction. A designer might select several components through the editor GUI and delete them all with a single command. The editor groups the deletions into a single transaction to report to analyzers. This avoids the need to report analysis updates relative to ephemeral states that are meaningless to the designer.

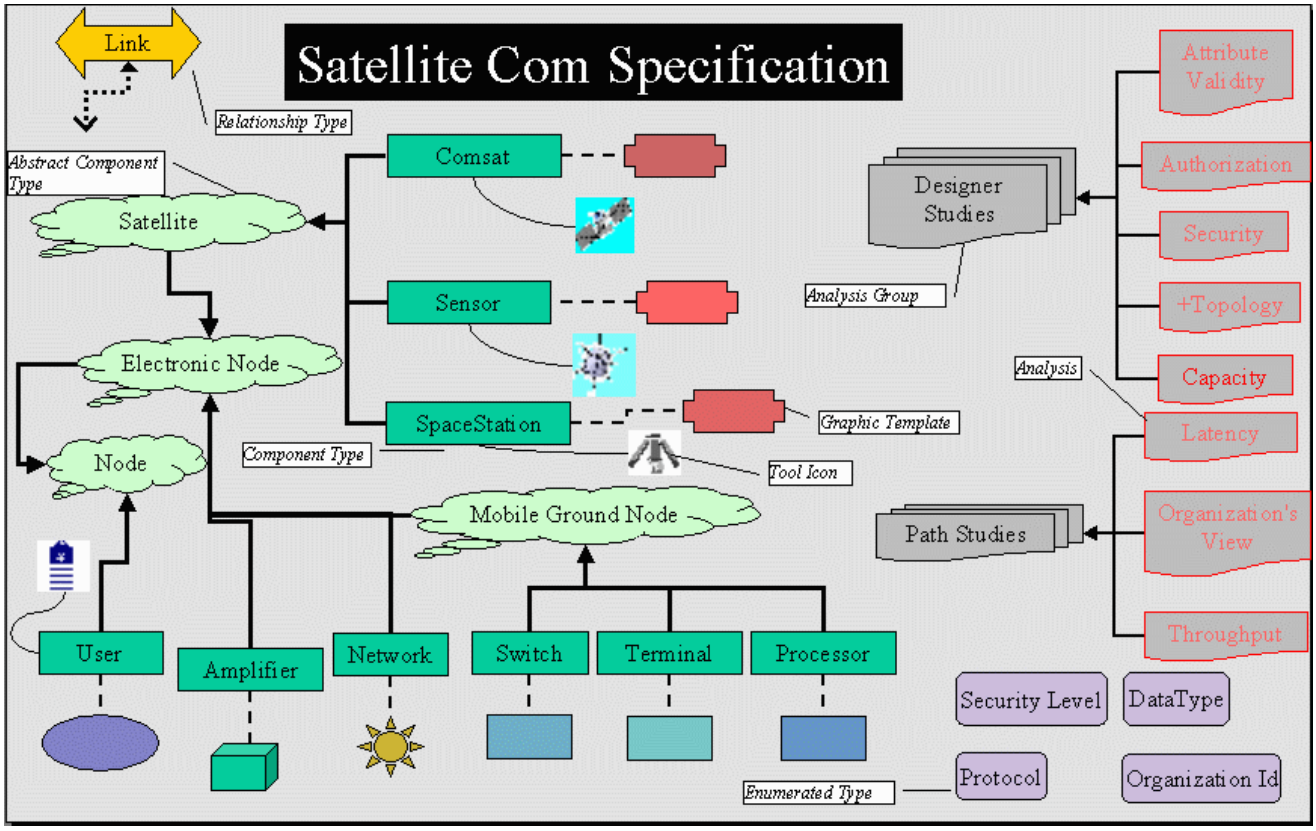


Figure 10: Domain Specification – Satellite Communication

Specifying New Domains and Generating Design Editors for them

The GUI described below in the Design Editor GUI section contains no novel features. We wish to reiterate that there is only a superficial understanding of the domain represented in the GUI itself, excluding the *content* of the analysis results. The novelty comes from two sources, the first of which is the means used to generate that GUI. The second, extending a widely used COTS product, is discussed in the next section.

The "Satellite Communications" GUI was generated with no traditional programming. Its specification, created through another graphic interface, is shown in **Figure 10**. The (green) rectangles labeled "Comsat", "Sensor", "User", etc. determine the domain's component types. The cross shapes attached to them by dashed connections are their *graphic templates*. This determines the appearance of an instance of the type when an engineer instantiates it in a design. The GUI designer either chooses a graphic template from a large library of shapes, or may import an image, in any of a variety of image formats, as a graphic template. The GUI designer tailors the template's color, border, and label text in this graphic domain specification.

A type specified may be connected (via a curved solid connector) to an image that serves as the *tool icon* for the type in the generated domain toolbar. Tool icons, like graphic templates, may be selected from a shape library or use an imported image. If no tool icon is specified, a scaled version of the graphic template is used as the tool icon.

The (gold) arrow shape labeled "Link" provides the sole relationship type in this domain. The dashed, double-headed arrow attached to it is the graphic template for the "Link" relationship type. The GUI designer tailors the color, dashing and arrowhead styles of a

relationship template in the graphic domain specification just as he tailors component type templates.

Single-inheritance hierarchies of unit types can be specified by placing abstract types, such as “Satellite”, in the design. Properties can be associated with either abstract or unit types. Property definitions are entered through a dialog like the one in **Figure 11**. A specification consists of a name, a type selected from a drop-down list, optional upper/lower bounds for numeric types, required/multiple indications, and a textual explanation. The explanation will appear in a small pop-up window when the designer hovers the mouse on the “tab” for that property in a property-editing dialog.

Any unit or relationship type may have initial property values specified through a property-editing dialog, identical to the ones used by designers. The default values are assigned when new instances of the type are created.

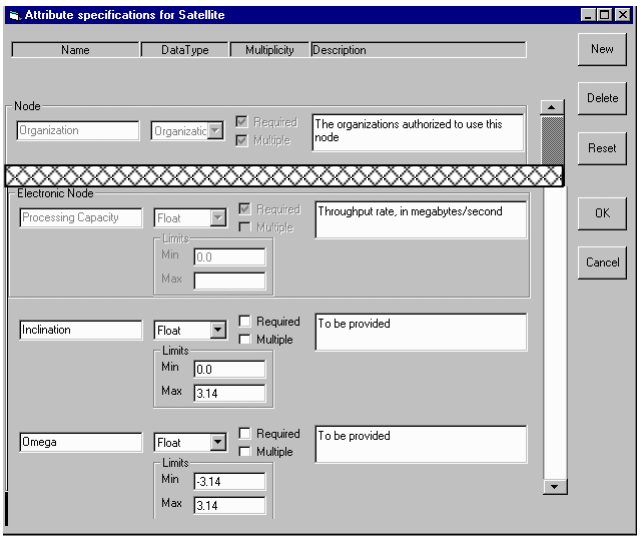


Figure 11: Property Specification Dialog

Figure 10 contains the specification of two global root analysis groups, “Designer Studies” and “Path Studies”, and eight analyses. The color and styling of the border of an analysis specify the means used to highlight components and relationships directed from markups in the feedback from the corresponding analyzers. Analogously, the text characteristics – font, face, size, color – of the label of an analysis specify the textual characteristics of any on-design markup text found in feedback from the analysis.

Generating Design Editors for New Domains

The specification-by-example editor is little more than a domain-specific editing environment specified with its own (partially bootstrapped) graphic domain specification for the “domain-definition domain”. A PowerPoint presentation file created by editing a design in the domain-definition domain serves as the specification for a new domain. Currently, the file name itself serves as the new domain’s name. When a designer begins editing a design for a domain D, D’s graphic specification is loaded in an invisible, read-only, mode into PowerPoint. The design editor then extends PowerPoint’s GUI by interpreting the content of that graphic domain specification.

We have implemented two “analyses” for the domain-definition domain. The first reports various errors such as unnamed types, circular inheritance, types without templates, etc.

The second “analysis” is a generator that produces an ASCII file containing definitions (in CommonLisp) for classes that correspond to those defined in the domain definition. A domain-independent CommonLisp module provides a mapping between this Object Oriented (OO) model and the editor-analyzer protocol. CommonLisp analyzers can then be implemented for this domain by programmers without any knowledge of DCOM and with all of the classes of that domain suitably defined.

Design Editor GUI

The central component of the design editor is its GUI. The editor’s *GUI* provides the interactive user with means to load/save designs, navigate within designs, create/delete/copy components and connectors, view and modify properties of components and connectors, and request analyses.

Figure 12 below is a screen shot of an editor generated for a “satellite communications” domain. Everything in the figure is part of the GUI with the exception of the callouts highlighting specific elements.

Readers familiar with Microsoft PowerPoint will immediately recognize many elements from that product’s GUI in this figure. This is discussed in detail in the Advantages of Extending PowerPoint section. Here we focus on the domain-specific aspects of the GUI.

In the central canvas is the design of a “satellite communications” configuration. The various labeled shapes represent instances of satellites, terminals, switches, processors, and users – the component types of the domain. They are connected by arrows representing communication links – the only relationship type used in this domain.

The designer created these design units through unit creation tools on the domain toolbar, seen near the upper right of the figure. To the immediate left of these tools is a drop-down list box displaying the name of the domain (“Satellite Com”). When a designer starts a new design, this box allows him to choose a domain. This triggers creation and display of the appropriate domain toolbar. Manipulation of units on the canvas – positioning, resizing, selecting, attaching/detaching links – is carried out through conventional mouse gestures and/or keyboard shortcuts.

The window displays a list of reports. In this example, there was just one report. Its explanation reads “User U3 is directly connected to user U2.” When the designer selects a report, its associated markup instructions are carried out. Their effect is reversed if the report is deselected, or the analysis window is closed. In this case, the only markup instruction called for highlighting the communication link between U2 and U3. That is why that link has an appearance (a thin red arrow) different from the others.

Property values are viewed and assigned through dialogs, displayed on demand from the unit context menus.

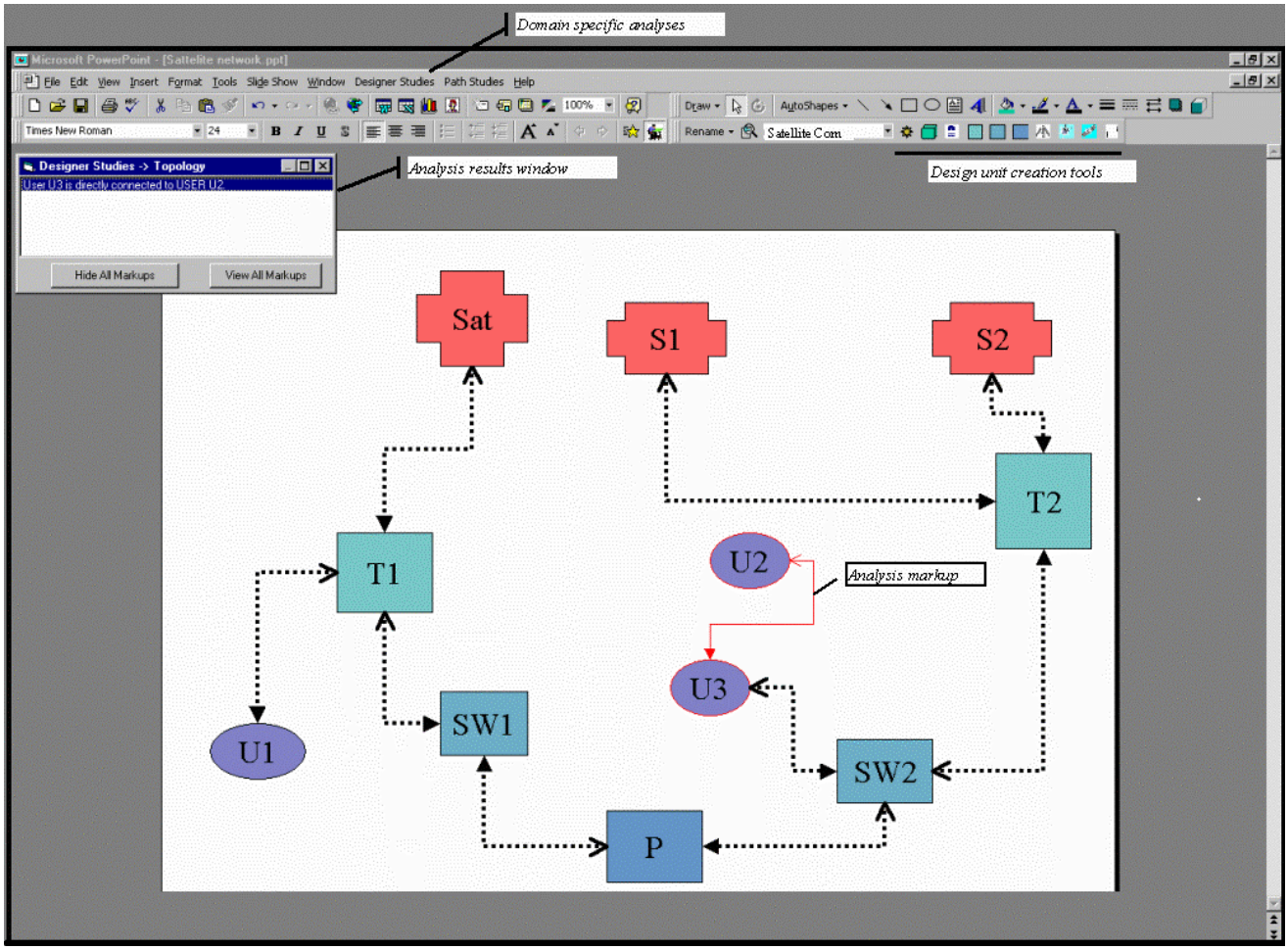


Figure 12: Design editing GUI – Satellite Communications Domain

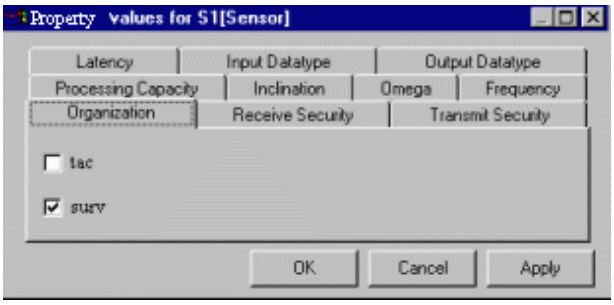


Figure 13. Property Value Dialog

Figure 13 exhibits the dialog for a sensor satellite. The dialog contains a "tab" for each property associated with the type in the domain specification. The details of a tab depend on the value type of its property and on of the domain specification. Identical dialogs are used to gather the parameter values for parameterized analyses.

Implementing the Design Environment with a COTS product

The design editor is implemented as an extension to Microsoft PowerPoint, programmed in Visual Basic. Technically, this extension is a COM server that receives "events" as the user modifies a design. The same module acts as a COM client of PowerPoint enabling it to navigate through a design and to paint analysis feedback directly onto the design. For efficiency reasons, this module runs as an "in-process" server. This means it is actually part of the PowerPoint process itself. Method calls are extremely fast when both client and server are part of a single operating system process. Greater efficiency could be achieved by implementing the extension in C++, but the performance of the Visual Basic code has been acceptable to us to date.

Design editor - analyzer protocol

When an analyzer process starts, it registers its interest in one or more domains, and registers as the provider of one or more analyses. As designs are loaded into the design editor, or modified, the design editor receives events from PowerPoint, interprets those events in terms of changes relevant to analyzers, and notifies registered analyzers.

When a designer requests an analysis and provides its parameters, the design editor notifies the registered analysis provider. That analyzer is subsequently expected to send the design editor an analysis. The design editor then presents the analysis to the designer.

Every update sent to an analyzer is marked with a monotonically increasing transaction count. When an analyzer reports an analysis, it includes the transaction count at which the analysis was computed. Any visible analysis based on a non-current design state is visibly marked as out-of-date by coloring the background of its report window.

When an analyzer has provided an incremental analysis, it is expected to update the analysis each time it receives a design update from the editor. When the designer closes an incremental analysis, the analyzer is notified and ceases to transmit updates.

The protocol allows an incremental analysis to be updated either by total replacement or by selective deletion and addition of reports

Communication between the design editor and analyzers takes place via distributed COM (DCOM). The rationale for choosing DCOM over, say, CORBA, to implement communication between the design editor and analyzers is only that Visual Basic, the language in which we implemented the design editor, trivializes the implementation of DCOM clients and servers. The fact that PowerPoint itself exposes (D)COM interfaces is *not* a factor, because currently the design editor does not pass analyzers direct references to any PowerPoint objects.

Although we have not done so, it would be reasonable to further categorize analyses as synchronous vs. asynchronous. Synchronous analyzers could use a simpler protocol (no need for transaction counts) and be allowed to run as "in-process" servers for high performance.

Advantages of Extending PowerPoint

PowerPoint is marketed as, and known to most of its users as, a presentation graphics editor. As such, it is viewed as an interactive editor of *presentations* consisting of multiple *slides.* However, it is also a *high level* graphic server, permitting independently written modules to read and update almost any aspect of its state and invoke numerous

methods through an object-oriented COM interface. But what does PowerPoint offer that is missing from traditional "visual interface" authoring tools?

Primarily, PowerPoint offers a highly functional GUI for interactively designing presentation graphics. Virtually every part of that GUI is useful, *without modification*, as part of our design editor. This includes:

- Scrolling, zooming, scaling, multi-slide designs
- Loading/Saving/AutoSaving designs, multiple windows, multiple views.
- Object deletion, selection, grouping, cut/copy/paste, and text formatting.
- Object positioning, alignment, rotation, reflection, resizing, graphic formatting.
- *Connectors* – self-routing lines/arrows whose ends attach to other objects, and adjust automatically to repositioning and resizing.

We emphasize that it is not simply the fact that PowerPoint has a library with methods for accomplishing these operations, but that it has a functional GUI that allows the designer to invoke them conveniently. If one thinks of an engineering design as a specialized PowerPoint presentation, it is not surprising that we have found no reason to *remove* any of PowerPoint's standard GUI. For example, any graphic object created through conventional PowerPoint tools may be placed in a design. Such *annotations* will persist with the design but will be invisible to analyzers – just as comments in programming languages are invisible to compilers.

One other feature of PowerPoint, though not part of its GUI, has also leveraged our implementation. PowerPoint allows arbitrary information to be associated with presentations, slides, and graphic objects in the form of string-valued tags. This is sufficient for the design editor's needs to store its own non-graphic design information, such as the property values that a designer has assigned to a unit. PowerPoint ensures that this information persists as part of the saved presentation document – no additional persistence mechanism had to be implemented for these extensions.

Finally, we note that the PowerPoint GUI is *already familiar* to many engineers, who use PowerPoint to present designs to clients and other engineers. In fact, some of them have commented that they have existing PowerPoint presentations they would like to *import* into our design editor.

Disadvantages of Extending PowerPoint

We should not give the impression that extending PowerPoint's GUI provides the same flexibility as building a hand-tailored design editor GUI.

The biggest impediment was the lack of "event" notifications in PowerPoint. Most of the design editor's activity must be triggered by some event in the GUI – or, more specifically, by a state change initiated from some GUI event. For our own GUI extensions (such as our dialog for editing unit attributes) there was no problem providing suitable notifications to the editor. However, detecting events initiated through the native PowerPoint GUI was a serious problem. Although a COM interface could make relevant events available, the interface implemented by PowerPoint97 *does not*. We developed a Design Monitor that employed two mechanisms to overcome this limitation.

The menu items and control buttons in the PowerPoint GUI are objects in the documented model. We found a way to replace them with equivalent ones whose reactions invoked our own code, which internally synchronously invokes the original

reaction. The mechanism is obscure, but relies only on documented operations and is fully general. For menu items, this method works independent of whether the invocation is by mouse or by keyboard shortcut.

However, “wrapping” the action associated with a command does not always provide an efficient means to determine the design-relevant events performed by the action. An extreme example is the “Undo” command. Although we may have control both before and after PowerPoint executes that action, we have no effective means, short of a complete comparison of before and after states, to determine the relevant state changes. The best we can do is simply remove such tools from the GUI, which is trivial. However, removal is undesirable, because the tool provides useful functionality for design editing, just as it does in editing presentation graphics. We have not found a satisfactory solution.

Events initiated by mouse clicks and motion within a design window were far more problematic. PowerPoint provides its extenders with no insight into those events or, more interestingly, into the changes to its state that result from handling them. Like any Windows program, mouse events are communicated to PowerPoint by the operating system through a message queue. Like other Windows programs, PowerPoint often responds to the lowest level mouse events by placing other, higher-level, events into its own message queue. Mechanisms independent of PowerPoint allow us to monitor messages being removed from this queue. Based on observations from a “message spy” program, we have developed ad-hoc rules to determine localized bounds (generally the currently “selected” units) for what *may* have changed in a design. We can then efficiently determine what design-relevant changes *actually* occurred by comparing our cached old state with PowerPoint’s current state within the affected locale. The fact that we are not concerned with most graphic details speeds up this comparison significantly.

Version considerations

Using a COTS product as a system component makes version upgrade concerns more significant than is the case with conventional runtime library components. How will a new version of PowerPoint impact the design editor? Because our designs and domain definitions are fully standard presentations, the new version will certainly automate any file format changes required to make them work. All our code that relies on the advertised (D)COM object model should require no change, because numerous other third party PowerPoint extensions rely on the same model. Existing menu items and controls in PowerPoint’s GUI might be removed, relocated, or renamed, but adapting to those changes would be trivial. New controls or menu items might appear, but our ability to wrap their actions simplifies dealing with them. However, because our rules for interpreting the significance of messages in the message queue are based only on *observation of the current version*, there is reason to expect they might have to be revised in potentially non-trivial ways.

3.3.5.2. Integrating EMACS as the Message Composition Editor for Eudora

By instrumenting the connector between the user interface (Windows) and the application, EMACS has been integrated as the editor for replying to messages in Eudora. When the reply button is clicked in Eudora, the body of the reply is sucked out of Eudora, placed in an EMACS buffer, and EMACS is switched to as the current window. When the composition of the body is complete, it is pushed back into Eudora,

the message is queued for delivery, and Eudora is switched to as the current window. Neither COTS tool was changed.

3.3.5.3. Personal Web Annotator (from Internet Explorer)

We developed a Personal Web Annotator that dynamically adds information to pages being downloaded to reflect local information. Annotations include New and Updated icons for links that are respectively new and modified since last read by the user (known by monitoring the user's browsing) and displaying the user's personal ratings for links previously visited.

3.3.5.4. Ad Buster (from Internet Explorer)

We also developed an Ad Buster for eliminating unwanted advertisements from downloaded web pages. This was accomplished by simply modifying the communication restrictions of our Safe Execution Environment for Web Browsing to prohibit communication with those sites hosting these advertisements.

4. Non-Bypassable Security Manager For Windows

We have explored how such architecture mediation, including requests for operating system services, can be used to enhance COTS products by transparently providing them with enriched services and by transparently integrating them into larger aggregations.

As Phase 1 of this effort we conducted a technical feasibility study to explore whether this same mediation technology could be used to create a non-bypassable Security Manager for the Windows NT operating system. That study concluded that a non-bypassable Security Manager for NT is feasible using our mediation technology called Instrumented Connectors.

Phases 2 and 3 of this effort were funded under this contract to create respectively a non-bypassable version of these Instrumented Connectors for the Windows NT operating system, and to use these non-bypassable Instrumented Connectors to construct a Security Manager for Windows NT. This Windows NT Security Manager will mediate requests for operating system services and interactions with other processes and determine whether to allow or prohibit those requests and interactions. This determination will be governed by a set of administrator supplied rules which define what programs are, and are not, allowed to do.

4.1. Scope

Programs interact with one another to obtain and provide services. By monitoring and mediating these interactions, a third party can control how those programs interact. It can prohibit particular interactions, change the services requested and/or provided, and alter the perceived environment in which one or more of the programs operate.

We developed the technology to encapsulate the execution of arbitrary programs so that they can be safely executed. This enables people to share and use applets, active controls, agents, and downloaded programs while ensuring the integrity and survivability of their information and computational resources.

4.2. Approach

We created a non-bypassable version of our Instrumented Connector technology so that when mediators are installed between a program and its services there is no way for that program to obtain those services without going through the installed mediators. This requires that the program can neither undo the installation of the mediators nor create another path to the services that does not go through the installed mediators.

In addition, the installation of these mediators must occur at the very beginning of the execution of the program so that it is unable to obtain any of the "mediated" services before those mediators are installed. Finally, the program must not be able to prevent the installation of the mediators. That choice is the provenance of the separate Windows NT Security Manager constructed under Phase 3 of this effort (and under this contract).

4.3. Objectives

ISI will build a mechanical transformer to modify binary versions of the Windows NT operating system (i.e. no source code is required) to incorporate non-bypassable mediators by placing the mediators themselves into the operating system so that service requests necessarily pass through these mediators.

ISI will build a mechanism to deploy the mediators for a program before that program starts to execute so that the program's entire behavior is mediated.

ISI will build a Security Manager for NT that installs these non-bypassable Instrumented Connector wrappers on newly spawned processes as they are started in accordance with a policy specification it is given. This policy specification will detail which wrappers are to be placed on which processes whenever they run.

The Security Manager will also propagate all wrappers installed on a process to all processes that it spawns unless explicitly overwritten in the policy specification.

To facilitate the management of wrapper installation and policy, ISI will develop a wrapper installation database that contains the set of installed process wrappings (i.e. each installation of a wrapper on a running process). ISI will also develop a wrapper policy database that contains the set of wrappings specified for newly spawned processes in the security policy specification. Both databases will have a query API that allows the contents of these databases to be dynamically examined.

ISI will also provide an API for dynamically changing the security policy for which wrappers to install on newly spawned processes. The API will allow these changes to either be made globally or localized to the (future) offsprings of one or more existing processes.

4.4. Accomplishments

We implemented non-bypassable mediators so that even malicious programs can neither remove nor bypass the installed mediators. This non-bypassability enables suspect and/or malicious programs to be sandboxed to protect sensitive information from being accessed or modified and other program executions from being disrupted. DARPA's ISO Information Assurance program funded the implementation of this capability.

We built a Security Manager for NT that installs non-bypassable Instrumented Connector wrappers on newly spawned processes as they are started in accordance with a policy

specification it is given. This policy specification details which wrappers are to be placed on which processes whenever they run. The Security Manager also propagates all wrappers installed on a process to all processes that it spawns unless explicitly overwritten in the policy specification. DARPA's ISO Information Assurance program funded the implementation of this capability.

We developed a safe execution environment for Web browsing. This execution environment ensures the safe execution of both Netscape Communicator and Microsoft Internet Explorer running with arbitrary Java and ActiveX applets so the full power and interactivity of Web material can be utilized. User specified rules restrict which files can be read and written, which parts of the registry can be read and written, which remote sites can be communicated with for uploading and downloading information, and which processes can be spawned.

We developed a safe execution environment for running office products. This execution environment ensures the safe execution of Microsoft office products even when they load documents with arbitrary macros. It uses the same set of meditors as employed by the Safe Web Browser to restrict the behavior of possibly malicious macros. However, the set of rules followed by these mediators differs and is tuned to the needs of the specific office product (e.g. it prohibits modification of templates to prevent macro propagation).

4.4.1. Red Team Experiments

4.4.1.1. Protected Path Experiment

In support of our task to increase the robustness and survivability of Instrumented Connectors, we've designed and are participating in a Red-Team experiment under ISO's IA program to determine whether Instrumented Connectors can establish protected paths on the Windows NT platform.

This experiment tests whether such protected paths can be established between:

1. a smart card and an application (Netscape) that obtains certificates from the smart card
2. the keyboard and the smart card that obtains the user's PIN to unlock the data contained on the smart card.

In this experiment, we attempted to use our Instrumented Connectors to keep the information passing through these protected paths from being snooped or tampered with.

To test the strength of these protections, the Red Team was given Administrator/Root privileges and the ability to run any programs they want.

Results

Using a variety of hacker tools downloaded from the web, the Red Team was unable to snoop on or tamper with either of the protected paths established for the experiment.

However, with access to the source code for the wrapper defenses and knowledge of how they worked, the Red Team was able to discover an unmediated (i.e. unprotected) NT interface that gave them access to keyboard data. Using this unprotected NT

interface they were able to write a program that snooped the key entered by the user through the keyboard.

Lessons Learned

1. Wrappers appear to be able to mediate selected NT interfaces and limit the usage of those interfaces to protect the resources controlled and accessed by those interfaces.
2. The size and complexity of the NT interface makes it difficult to ensure that all interfaces that need to be mediated to protect particular resources have been identified.

4.4.1.2. Hardened Client Experiment

As part of DARPA's OPX program, our SafeEmailAttachments wrapper was integrated by BBN with SCC's Autonomous Distributed Firewall and a commercial file encryption package to produce a "Hardened Client."

This Hardened Client was subjected to a Red Team attack (by the SRI Red Team) to test the strength of the defenses. Flags were placed to force the Red Team to attack each of these technologies.

Results

For our SafeEmailAttachments wrapper, the Red Team tried a variety of attacks using different types of attachments and message scripts. Most of them were blocked by the wrapper defenses, but two of them succeeded.

The first utilized a known vulnerability to tunnel below the wrapper defenses and remove them. At the time of the experiment DARPA had already funded an effort to fix this vulnerability, but the existence of this known vulnerability and its planned repair wasn't communicated to the Red Team. So they independently "discovered" and exploited this vulnerability.

The second successful attack resulted from a coding error in the defenses that failed to canonicalize a security rule before installing it. This allowed the resource that the rule was supposed to be protecting to be accessed and used to mount a multi-stage attack. Once this rule was canonicalized, this attack was blocked by the SafeEmail wrapper.

Lessons Learned

While the vast majority of the Red Team effort was devoted to the first attack, very little was learned since this exploited a vulnerability known to the defenders.

On the other hand, the success of the other attack was a surprise to both the attackers and the defenders, as they both expected the defense to block the transfer of data between the attack's multiple stages. The success of the attack forced us (the defenders) to track down the cause of the failure to block this data flow. In doing so, we discovered several vulnerabilities in the defense that could have been exploited (but weren't) before we uncovered the real reason for the failure (not canonicalizing the rule).

4.4.1.3. Non-Bypassability Experiment

DARPA has asked Sandia to conduct an independent assessment of our SafeEmail system (SafeEmail utilizes our Instrumented Connector technology but was developed under a separate DARPA contract), including its NonBypassability Protection.

We prepared documentation for Sandia to conduct this assessment, briefed them on the system's concept of operations, and answered questions on the system's operation.

Results

They discovered a vulnerability in our NonBypassability Protection that enabled them to defeat the protection it affords. This vulnerability utilized an error handler to field the error that arose from transferring control directly to the non-forgeable instruction signaling the kernel driver that trusted code had been entered (rather than entering this code through its normal entry point). This instruction was executed, putting the kernel driver into its trusted mode, and then when execution continued with the following instruction the error it caused (because the registers were purposely missed) transferred control to the attacker's error handler.

We designed a response to this class of attack in which the kernel driver disables any error handlers in the calling process before returning control to that process. Any error arising in the code following the non-forgeable instruction now causes the process to be aborted rather than allowing the attacker to gain control in the trusted mode. Before the trusted code returns control to the caller of a mediated API, it restores any error handlers that existed before the call on that mediated API.

5. Published Papers

The following selected set of attached papers that were published during this contract document the work performed on the various tasks and describe the accomplishments achieved:

1. Balzer, R., Instrumenting, Monitoring, & Debugging Software Architectures
2. Goldman, N. M. and Balzer, R., The ISI Visual Design Editor Generator
3. Balzer, R. and Goldman, N. M., Mediating Connectors
4. Balzer, R. and Goldman, N. M., A COTS Based Design Editor with User-Specified Semantics
5. Goldman, N. M., Smiley – An Interactive Tool for Monitoring Inter-Module Function Calls